DataLab: A Version Data Management and Analytics System

Yang Zhang
Institute of Interdisciplinary Information Sciences, Tsinghua University
zhangyang14@mails.tsinghua.edu.cn

Fangzhou Xu
Institute of Interdisciplinary Information Sciences, Tsinghua University
xfz@mails.tsinghua.edu.cn

Erwin Frise
Berkeley Drosophila Genome Project, Environmental, Genomics and Systems Biology Division, Lawrence Berkeley National Laboratory, Berkeley
erwin@fruitfly.org

Siqi Wu
Department of Statistics, University of California, Berkeley
siqi@stat.berkeley.edu

Bin Yu
Department of Statistics & EECS, University of California, Berkeley
binyu@stat.berkeley.edu

Wei Xu
Institute of Interdisciplinary Information Sciences, Tsinghua University
weixu@mail.tsinghua.edu.cn

ABSTRACT

One challenge in big data analytics is the lack of tools to manage the complex interactions among code, data and parameters, especially in the common situation where all these factors can change a lot. We present our preliminary experience with DataLab, a system we build to manage the big data workflow. DataLab improves big data analytical workflow in several novel ways. 1) DataLab manages the revision of both code and data in a coherent system, and includes a distributed code execution engine to run users' code; 2) DataLab keeps track of all the data analytics results in a data work flow graph, and is able to compare the code / results between any two versions, making it easier for users to intuitively see the results of their code change; 3) DataLab provides an efficient data management system to separate data from their metadata, allowing efficient preprocessing filters; and 4) DataLab provides a common API so people can build different applications on top of it. We also present our experience of applying a DataLab prototype in a real bioinformatics application.

1. INTRODUCTION

People have generated large volumes of data in recent years. Although "data scientist" has become a hot position in many companies, there are few tools to help their workflow. As the data science code becomes more complex, many data analysts start to adopt code revision tools such as Git. However, the demand of data science goes beyond what Git can provide.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

BIGDSE'16, May 16 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4152-3/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2896825.2896830

First, data science is data-centric, and a dataset can go through multiple stages of cleaning, labeling and other preprocessing steps. There might be different versions of each step. The data scientists need to track the process and the revisions. A common (but bad) practice is to make redundant copies of the datasets and get confused by meaningless file names such as `data.csv`, `data-version1.csv`, `data-final-version.csv`, `data-last-version.csv`. Our experience shows that using the wrong datasets or wrong versions has been a common source of bugs.

Second, there are many parameters in a machine learning model, and tuning them to fit the data is a common task in data science. The huge number of parameters such as learning rate, initial values, regularization values and so forth often confuses people. As a result, they can easily lose the intuition of significance in these parameters to the result.

Third, when the datasets get larger, it is often necessary for data scientists to setup a distributed platform, and redo the entire experiments management process in a distributed system. They are likely to adopt many third party packages and tools in a data science project but unfortunately it is usually tedious to configure these packages on machines with different hardware/software environments.

Last but not least, it is difficult for data scientists to share their experience about the datasets. Of course they can share their code and sometimes the analysis results, but it is hard for other people without a deep understanding of the particular dataset to take advantage of the code or the results as they are highly tailored to the data.

The MIT DataHub [1] [2] project supports dataset revision control, but it does not manage the entire data analytics development cycle. Thus, it is more of a database management tool than a software engineering tool. On the other hand, the Harvard Dataverse is a data publishing and sharing platform but it lacks data version control and analytics.

In this paper, we present our early experience on DataLab, a new tool that integrates code, data and execution management into a single system. DataLab provides a simple API so that different types of users can build data specific tools

and interfaces on top of it. There are three key ideas in DataLab:

1) DataLab keeps both data and code versioned. Each version of the code has the corresponding version of data. The only way to modify a dataset is through executing some code (and we encode the rare events of manually editing on the data directly as a special type of modification code). With this design, the users can inherit experiences from code revision management and extend it to data management in a natural way.

2) For efficiency reasons, DataLab automatically stores and manages data and metadata separately. Metadata are fields that are commonly used by the users when they filter or display the datasets. For example in ImageNet [6], raw data are the image files and metadata contains the corresponding file name and classification labels. During the preprocessing steps, users often perform selections based on the metadata, and this organization provides efficient selection operations. The data and metadata are linked by unique IDs.

3) DataLab includes data processing engines not just as a user convenience tool, but also it enables automatical recreation of datasets that is no longer available in the storage. DataLab keeps all versioned data and codes as a directed acyclic graph (DAG), which we call *data work flow* (DWF), to remember the dependency among these entities. In this way, we can reproduce any deterministic executions at any time as long as we have the raw data and the code.

There are five tightly coupled key components in DataLab. They work together to solve the four problems above. *The data manager* manages the data and metadata. *The code manager* extends the GitLab [8] system and connects codes with the datasets using unique submitting IDs. *The execution engine* integrates multiple processing backends, including both single node engines like R or Python and distributed frameworks like Spark [24]. *The user interface module* provides comparison tools on both code and data, as well as an online editor. Our experience shows that the interface module is essential for small edits to the code, especially for non-technical users. Finally, the *system core* maintains the data work flow graph model and schedules all the dataset caching / executions. We provide an intuitive API to access all the DataLab functionalities.

We have implemented an early prototype of DataLab and developed a bioinformatics application with real users based on the DataLab API. The prototype supports multiple data formats with versioning and provides both single machine and Spark-based execution engines. With DataLab, we solved many experiment and data management problems our users face. We describe the early experience developing these functionalities in Section 3.

2. DATA MODEL

We introduce the DataLab data model in this section. We first describe the logical data model that presents the user-facing data structures, including code, data and execution records. Then we discuss about how we link all these data together using a data work flow graph, and it is the core functionality for DataLab system. Finally we briefly introduce how we manage the logical data structures efficiently on a distributed cluster with a combination of revision control systems, NoSQL databases and distributed file systems.

2.1 Logical data model

DataLab manages three types of "data", the dataset under study, the analytics code and the system execution records. We first introduce the data model from a user point of view and then we discuss how we implement the physical model efficiently in Section III.

Logical Dataset.

From a user's point of view, the data under study are organized as semistructured data tables. The users can issue queries over the table. The users can only query the metadata fields. The result of the query is either a data table containing only the metadata fields, or a data file with both the metadata fields and the raw data (such as the images).

We allow the users to add or modify the metadata fields. For example, when a user generates a new set of labels for each image in ImageNet [6] either using a machine learning algorithm or manual labelling, she can choose to introduce a new field with her own label, or modify the existing field containing the label. All the user-created metadata fields are treated the same as the first class citizen and can be queried in the same way as all other fields. The new field is versioned, and all versions are always accessible to the users. The users can specify which version of the fields they want to query.

Users can create new datasets too. Users can also name a new dataset. This is a common practice for data preprocessing tasks - the users can preprocess the dataset into a new one and start more analytics from the preprocessing results. In this case, a convenient name for the resulting dataset can be handy. DataLab remembers how each dataset is derived from other datasets and code, and we will discuss more about the dependency tracking in the next subsection.

Code.

In DataLab, users' code is the key to the entire data analytics process. Users' code includes two parts, the program texts and the configuration parameters. We adapt GitLab to manage both so users can get a single consistent interface. We use the Git commit ID as the unique IDs for each version of both the code and the data. We extend the GitLab API allowing users to compute the resulting dataset by executing the code.

The current configuration file is a key-value storage that is available for both DataLab system and the user's program. The users can specify the input dataset, the output dataset / field of the program, as well as which programs / platforms to run on. The DataLab system takes these parameters and runs the program.

The users can also specify some variables in the configuration files for their own code, such as various parameters for machine learning algorithms. These parameters are made available in their programs as if they are passed in through the command line. Forcing users to specify the parameters in a configuration file rather than real command line helps DataLab to keep track of the entire analytics process, minimizing chances that users make untracked parameter changes.

System Execution Records.

Internally in DataLab, we keep records for each step of the users' code submissions and executions. After receiving

user's submission request and executing user code on the DataLab system, we automatically store the commit ID, the commit time, execution logs and so on. These execution records are essential to maintain the data work flow we will discuss next.

2.2 Data Work Flow

To maintain the version hierarchy of datasets, we proposed the concept of data work flow (DWF) as the logical relations behind DataLab system, with which datasets are connected to each other based on their derivation dependencies. DWF is the core of DataLab and the foundation of reproducible data management.

In a DWF, a node represents a particular version of a dataset. A directed edge connects two nodes if one dataset is derived from the other. The labels on the edges show the code version that is used in the experiment. Figure 1 provides an example of the DWF.

It is obvious that the DWF graph is a directed acyclic graph (DAG). Figure 1 shows two common structures in DWF: one-to-one and many-to-one.

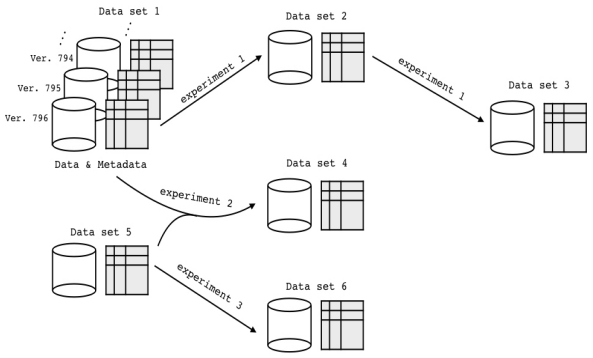


Figure 1: An example of DWF graph

In a one-to-one structure, a dataset is derived from a single parent dataset. For example, users can create an annotated dataset and link it back to the original dataset, and they can also generate new datasets for further research and share them with other users. Many-to-one structure represents the case that one dataset is derived from two or more parent datasets. Many operations like merging data from two spreadsheets into a single table.

We implement these structures by adding a *parent* property to the metadata of the new generated dataset so that when we process one dataset we can find the parent dataset by reading this property directly. Also, we implement functions to compare the differences between a dataset or programs to its parent or child dataset. This feature helps users to easily see the consequences of their code changes.

The DWF graph serves two purposes: 1) it is used to allow the system to schedule the evaluations / re-evaluations of certain datasets, and 2) it allows users to manage their entire experiment history, including knowing which *version* of the dataset generates which results.

DWF is similar to the concept of data dependency graph that is common in many systems. For example, Spark uses a directed acyclic graph to manage Resilient Distributed Datasets (RDD) [23] while Dryad [9] uses the dependency graph to organize individual partitions and steps for distributed computing. We use the graph differently in that DWF keeps track of the execution history and versions of the datasets, rather than the intermediate execution states.

2.3 Physical data model

It is non-trivial to implement the logical data model efficiently, as naively we have to keep all existing versions in the storage and manage a large number of datasets. Here we introduce the general ideas of our physical data structures.

Physical Datasets.

First, we separate the storage of data and metadata. For example, the former can be a bunch of image files, video files or system log files, which are usually too large to store in a database, and thus we store them in a distributed file system like Hadoop File System (HDFS) [18]. With modern distributed file systems, we can sequentially scan on the datasets efficiently but it is impossible to perform efficient random access into a particular data point. To solve the problem, we store the labels and annotations of each image, such as file name, size and content descriptions separately in a NoSQL database to accelerate queries. We link the data and metadata with a system-generated ID.

As we keep versioned data for each field, a new dataset is just a collection of fields with an explicit dataset name and version number. Currently we use MongoDB to store all the data, and as an on-going work, we are migrating to a column-oriented database to further improve performance.

Managing code revisions and the data work flow graph.

In the physical point of view, users' personal information, submitting records and corresponding parameters are stored in our Database system.

We store all users' codes in a GitLab server, and we use the GitLab API to communicate with it. All the execution records, except for logs from the users' program, are imported back into the MongoDB as a special system table. The execution engine framework is mapped to users' project directories to simplify the path management for each user. We are migrating to a container-based deployment management system [14] as a future work.

We keep the DWF graph as a separate background task. As we keep track of all execution records, we are able to construct the DWF, which helps users to find out data provenance and understand their data better.

3. SYSTEM IMPLEMENTATION AND APIS

In this section, we present the physical architecture design and the details on how DataLab works internally.

3.1 System Architecture

Figure 2 shows the system architecture design of DataLab system that consists of the following five components.

The data manager.

manages the data and the metadata. For efficiency reasons, although logically we treat data and metadata as a single entity, physically we separate the storage as we describe in Section 2.

The code manager.

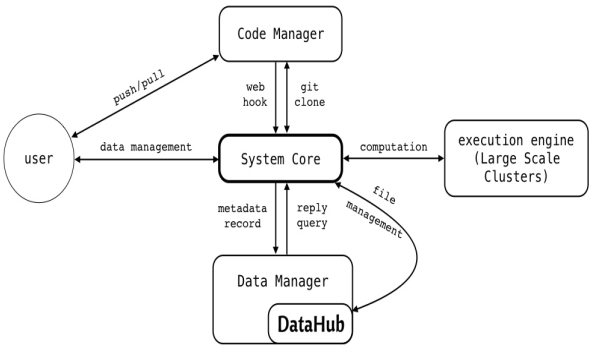


Figure 2: System overview

extends the GitLab system and connects codes with the datasets using unique submitting IDs. The code manager also manages a configuration file in which the users can specify the source / target datasets as well as various configuration parameters. The code manager triggers data processing execution and automatically reports the version of code / data being used.

The execution engine.

supports multiple execution backends, which can be one single machine or a cluster. Currently we support Python and R for single node, and Spark for clusters. The execution engine is a necessary part of the system for two reasons: 1) it is convenient for the users as setting up the distributed infrastructure would be tedious; and 2) we need automatic execution to link the code with its resulting datasets. In other words, executing the code within the DataLab system allows us to recover lost/deleted intermediate datasets as long as we have the raw data and the code itself.

The user interface module.

It is a common scenario that users analyze some datasets and make reports about the data, such as calculating precision and recall in a natural language processing task, or evaluating the daily investment returns in a stock market back-testing task. DataLab system is capable of comparing any pair of these analysis histories and displays the changes on codes or parameters, which lead to changes in results. This UI design helps users obtain best algorithm and parameter settings.

The system core.

It is the key component in DataLab that manages the relationships of all the executions and versions. It is responsible for processing user requests and managing resources for both computing and storage tasks.

We build our DataLab system based on open source projects including GitLab, MongoDB, Hadoop [22] and Spark. We reuse the UI of GitLab for code editing and DataHub for data display. DataLab makes full use of advantages of these projects and integrated them to provide stable services for users to manage their data and codes for data science tasks.

3.2 A typical workflow in DataLab

Data import.

Importing data is a prerequisite step for system core to engage computing. It also tells system core how to extract metadata from the raw data. For example, if user has a large set of images or unstructured logs, she needs to define the rule about how these data files are organized.

Code submission and execution.

Whenever a user pushes her code to GitLab server, the GitLab server notifies the system core through a web hook. System core pushes user requests to its queue, and at the same time, it pops the request at the head of the queue and handles it. System core copies the code in this request to the execution engine, which runs it with the user specified parameters and inputs. After the task is finished, the system core records this request, including the commit ID of this push operation of GitLab server, user specified parameters and any other application specific information. In occasions when runs of experiments generate new datasets, system core also records the relations between these datasets, (details in the next section). Notice that we copy only code, not data in the entire process.

Reading the results.

Finally, through the system core, a user can check the records of all her datasets and the entire execution. It would be very helpful for researchers to find the most suitable parameter value if they can compare all the results that she obtains alongside changes of parameters. Further more, users can share their datasets and analytic code with others by simply sharing the name of dataset and the commit ID.

3.3 Core APIs

DataLab provides API extending a normal code revision control system like Git, and Table 1 show the most important ones. Users are also able to extend APIs with their own codes, which will be described in Section 4 in detail.

4. CASE STUDY

We have tested a DataLab prototype in a real-world bioinformatics case study, and we report our early experience here.

4.1 Background of the application.

Many life science research projects depend on a vast amounts of data. We worked with the Berkeley Drosophila Genome Project [21] and the Department of Statistics at UC Berkeley to create a prototype application. The ultimate goal of the project is to identify gene interaction networks during development of the model organism *Drosophila melanogaster* (fruit fly) to infer putative genes involved in human developmental diseases or cancer. The project team consists of a group of biologists and statisticians.

The team has collected over 120,000 images showing spatial gene expression patterns of *Drosophila* embryos at different development stages. The dataset is publicly available at http://insitu.fruitfly.org. In an ongoing process, the team has been mining these images to study interactions among different genes. Co-occurrence of two genes at the same spatial location indicates gene-gene interaction but spatial patterns vary widely. They developed an analytics pipeline consisting of multiple preprocessing steps including align-

Core APIs			
name	functionality	input	output
create	create a project	dataset name	null
upload	upload a dataset to a project	file or directory name	null
import	import a dataset to the system	file or directory name	null
merge	merge two dataset into one	two dataset names	null
diff	check out the difference	two dataset names	spreadsheet
submit	push codes to system and execute	commit ID	null

Table 1: Core APIs

ing, resizing and registration. After preprocessing, several machine learning algorithms were applied to the dataset.

For example, to find common spatial regions in the gene expression patterns, the team developed a Nonnegative Matrix Factorization (NMF) algorithm to extract these regions, called principle patterns, from the image dataset with software package called SPAMS (SPArse Modeling Software) [13].

We used DataLab to support the data analytics. We provided two execution backends, Python and a Spark cluster. We support Python for compatibility with existing code, while Spark provides a distributed execution environment. We used DataLab to identify the principal patterns.

In the remaining part of this section, we summarize the problems that DataLab has helped to solve in the process.

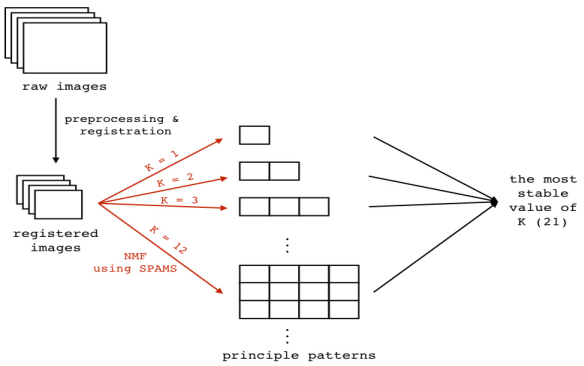


Figure 3: Pipeline of fruit fly image analysis case study

4.2 Preprocessing and data selection.

Users often want to choose only a subset of images. For example, they want to select only a few genes corresponding to certain annotation terms or based on previous machine learning results. With DataLab, the users can query the metadata with some codes such as selecting operations and automatically generate a subset of the images. This is the key benefit of separating the metadata with the raw data, and it helps save huge space in storage consumption and time in data transmission.

DataLab provides APIs to organize the data into a queryable format. Currently DataLab allows importing data from image files, SQL databases, `tsv` (tab separated) files, and `csv` (comma separated) files. Using the APIs, users can conveniently choose which columns to use as metadata or data, as well as filter out redundant or unnecessary data.

We customized the standard scripts of DataLab to provide dedicated filters on attributes such as gene names, related genes and data source. We also developed data importers to import datasets with metadata is embedded in the dataset itself, such as `DICOM` files, which are widely used in scientific imaging.

4.3 Supporting different preprocessing methods

Different analysis methods require different data preprocessing methods.

Sometimes the preprocessing steps create extra metadata that are required in the filters in later stages, in addition to the data itself. For example, the team developed an organ detection technique to segment later-stage embryo images showing the boundaries of the developing organs into areas with organ structures. In the preprocessing step, images are segmented into "super pixels" [15]. To classify the super pixel into organs, the team wants to label and query these super pixels. We designed a custom data structure to store the super pixel information as metadata. As the DataLab API allows for storing metadata in many different formats, we can easily store and manage this new data structure, just as the other metadata.

4.4 Managing parameters in different experiments.

The number of the principle patterns K is the key parameter for the NMF algorithm. To find the optimal number of K, the team proposed a novel model selection method, staNMF, to identify a K with the most stable NMF results from different initializations [19]. Figure 3 shows the entire pipeline. The stability measurement determined K by systematically setting K to different values, applying NMF with random initial values and measuring the stability of the output set of principal patterns. StaNMF selects the K with the highest overall stability. For each K, it takes 100 NMF runs for precise measurement of stability.

DataLab helps to keep track of the entire history for different results with various K values. Thus, the users can not only see the stability measurements of using different K values, but also the NMF results for using a particular K.

The speed of the NMF algorithm is limited. As datasets grow in size, the running time required to find the proper K and corresponding principle patterns will become unacceptable. Thus, the next problem that DataLab needed to solve is execution efficiency.

4.5 Handling large datasets efficiently.

As the image dataset gets large, the efficiency in processing directly affects the overall progress of the data analysis. For example, the preprocessed dataset contains a matrix of $405 \times 1640 = 664,200$ values. The final stage of the pipeline,

the NMF algorithm, takes 200 seconds for 10 repetitions setting $K = 21$ on a single core machine. We built two Spark clusters to increase NMF execution speed. The smaller one consists of 3 machines with 16 cores and 32G memory each while the larger one consists of 20 machines with 8 cores and 16G memory each. With these two clusters, same experiment takes only 20 seconds on the smaller cluster and 16 second on the larger one to finish, suggesting that expanding the cluster does provide more computation power.

4.6 Reviewing and comparing results under different setups.

In a traditional data analytics pipeline, as the data undergoes multiple stages of preprocessing and feature extraction, it is tedious to match which part of the results is derived from which part of the dataset, making it less intuitive for the domain experts to understand the result.

DataLab automatically generates a UI of the dataset after any execution. Specifically, it chooses a set of random samples to display on the UI. Users can choose to display these data samples side-by-side with those from the previous version of the experiment and can thus get a clear view of how their datasets evolves through each version.

4.7 Sharing the analysis results

The data analytics results eventually become part of the public dataset and the users want to store them in the long term and share these results publicly with other researchers. Current implementation of DataLab still lacks user permission management functionality, but as the data are all public, it is still useful in our case. We allow creating separate views in the UI to show some experiments and the results while hiding others. Also we allow users to name certain dataset with human friendly names. These features help sharing the results. We are working on user permission management subsystem as an important future work.

5. RELATED WORK

Source code version control and management system has a long history, from CVS to SVN to Git. These tools track the versions of files and keeps the different branches so that users can look back their files easily. They are designed to handle modest-sized files and thus not suitable for storing data.

Versioning has been a hot topic in database research field in recent works likes arrays [17] and graphs [11]. These work implement elementary operations for comparing differences between versions. Other temporal database systems like [16, 20] provide querying versions with linear chains but do not support other complicated structures of data versions like tree-structured versions.

Some workflow tools like Chimera [7], Pegasus [5] and Vistrails [3] take the concept of data workflow similar to the workflow we use in our DataLab system. However, they all lack the separation of raw data, metadata and versions of datasets. Other tools like Orchestra [10] and Fusion tables[12] use the concept of data collaboration among users but lack the capability of data version control and management.

DataHub [1] is built based on the systems above and it implements more sophisticated techniques for data version control and management so that it behaves like a Git service for data management. On the other hand, it also provides collaboration among users. However, users must manipulate their datasets on their own computers, which means they have to upload and download datasets frequently.

6. FUTURE WORK

Our early experience with DataLab reveals a number of important problems that we will focus on as our future work.

Storing all revision history efficiently.

Obviously it is impossible and unnecessary to store all datasets created by users' programs. We realize that we can reproduce every dataset as long as its parent datasets in the DWF are available and the execution is deterministic. In this situation, we can delay the computation to generate (or regenerate) a dataset until the user asks to look at the results or a descendant of the result. It is similar to the concept of *lazy evaluation* in functional programming. We plan to add lazy evaluation to DataLab system to make it a data-driven, "smart" system that only does the necessary evaluations, and only keeps the most likely used result datasets.

One challenge is that a model can be generated from a randomized algorithm with random seeds. To make sure that an execution is repeatable, we have to record all random seeds for a deterministic replay, if requested by the users.

Multi-users and data security.

Currently DataLab is for a single user. There are two challenges to support a multi-tenant environment. First, the data privacy is important. We will need fine granularity of data access control to specify which user can read/modify which part of the data. Data versioning and user-generated metadata makes the access control model trickier than a typical multi-tenant database system. Second, we have to handle malicious programs submitted by the users. For example, a malicious or buggy user programs can use up all system resources or even delete useful datasets. We plan to leverage the existing visualization and container technologies to keep the user program in a sandbox, and leverage our versioned data storage to keep the data safe.

Big data on the user behavior.

With DataLab, users perform the entire analysis in our system. It provides us with an excellent opportunity to study users' submission records. We can not only use these records to predict which dataset the user will request next for performance optimizations, but also we can learn how different people (programmers, data scientists, domain experts) interact with code, data and the resulting model. In this way, we may be able to automatically provide suggestions/hints to users, especially those who are new to the data science field.

7. CONCLUSION

The challenge of software engineering in big data applications lies in the tightly integration of code, data. Also, how fast we can process the data is also a big concern, often affecting people's choices on different algorithms. Through our early experience with DataLab, we demonstrated the benefit to integrate code, data and the execution environment as a single system. Using the data work flow graph, we can reproduce every version of dataset, keep track of user's experiment records for data analysis tasks and maintain dependencies

between datasets at the same time. We also need to notice that big data applications are domain specific, and thus we provide simple yet powerful APIs for users to extend and customize in order to fit different application. Our experience with a real bioinformatics application shows promising results.

8. ACKNOWLEDGMENTS

This research is supported in part by the National Natural Science Foundation of China Grants 61033001, 61361136003, 61532001, China 1000 Talent Plan Grants, Tsinghua Initiative Research Program Grants 20151080475, a Microsoft Research Asia Collaborative Research Award, and a Google Faculty Research Award.

9. REFERENCES

[1] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798*, 2014.

[2] Anant Bhardwaj, Amol Deshpande, U Maryland UMD, Aaron Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. Collaborative data analytics with datahub.

[3] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[4] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[5] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR,09*, 2009.

[7] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc of Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 37–46. IEEE, 2002.

[8] GitLab. http://gitlab.com. Accessed September 1, 2015.

[9] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[10] Zachary G Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. Orchestra: Rapid, collaborative sharing of dynamic data. In *CIDR*, pages 107–118, 2005.

[11] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008. IEEE, 2013.

[12] Jayant Madhavan, Sreeram Balakrishnan, Kathryn Brisbin, Hector Gonzalez, Nitin Gupta, Alon Y Halevy, Karen Jacqmin-Adams, Heidi Lam, Anno Langen, Hongrae Lee, et al. Big data storytelling through interactive maps. *IEEE Data Eng. Bull.*, 35(2):46–54, 2012.

[13] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online learning for matrix factorization and sparse coding. *The Journal of Machine Learning Research*, 11:19–60, 2010.

[14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[15] Xiaofeng Ren and Jitendra Malik. Learning a classification model for segmentation. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 10–17. IEEE, 2003.

[16] Betty Salzberg and Vassilis J Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.

[17] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1013–1024. IEEE, 2012.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[19] Ann S. Hammonds Susan E. Celniker Bin Yu Siqi Wu, Antony Joseph and Erwin Frise. Stability driven nonnegative matrix factorization to interpret spatial gene expression and build local gene networks. Submitted.

[20] Richard T Snodgrass. *The TSQL2 temporal query language*, volume 330. Springer Science & Business Media, 2012.

[21] Pavel Tomancak, Benjamin P Berman, Amy Beaton, Richard Weiszmann, Elaine Kwan, Volker Hartenstein, Susan E Celniker, and Gerald M Rubin. Global analysis of patterns of gene expression during drosophila embryogenesis. *Genome biology*, 8(7):R145, 2007.

[22] Tom White. *Hadoop: The definitive guide.* "O'Reilly Media, Inc.", 2012.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.